OpenMP and Structured Parallelism

Software and Solutions Group

Intel Software College

Objectives

Upon completion of this module you will be able to implement data parallelism using OpenMP

Agenda

Parallel Computing

- Introduction
- Factors Affecting Correctness

An Introduction to OpenMP

Parallel Programming Tips and Techniques

Most Code Contains Some Parallelism

Task parallelism:

```
fluxx (fv, fx);
fluxy (fv, fy);
fluxz (fv, fz);
```

Data parallelism:

```
for (y = 0; y < nLines; y++)
{
    genLine (model, im[y]);
}
```

Data Parallelism

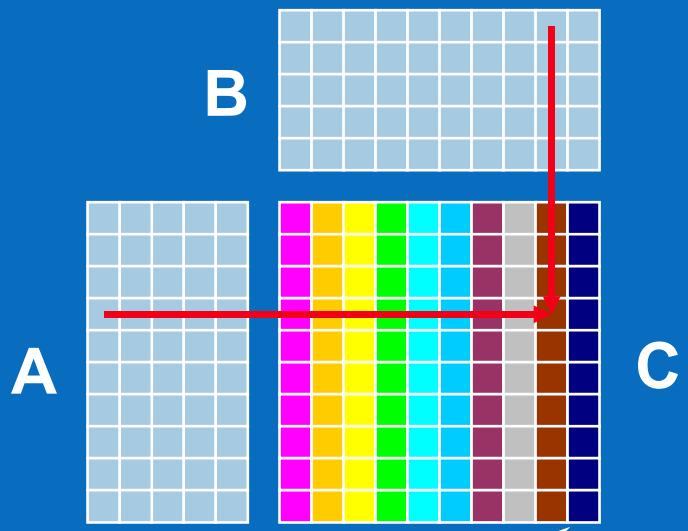
Same operation performed on different data

Requires data domains that can be computed concurrently

Normally found in loops

```
for (i = 0; i < M; i++)
   for (j = 0; j < N; j++)
      C[i][j] = 0.0;

for (i = 0; i < M; i++)
   for (k = 0; k < L; k++)
      for (j = 0; j < N; j++)
         C[i][j] += A[i][k] * B[k][j];
```

Example: Matrix Multiply




```
for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
        C[i][j] = 0.0;

for (i = 0; i < M; i++)
    for (k = 0; k < L; k++)
        for (j = 0; j < N; j++)
            C[i][j] +=
                A[i][k] * B[k][j];
```

Agenda

Parallel Computing

- Introduction
- Factors Affecting Correctness

An Introduction to OpenMP

Parallel Programming Tips and Techniques

What is Parallel?

Variables only read within a loop (fv,c,d)

Variables written then read in each iteration (f,cf2,dvf2)*

Variables only written but indexed on loop counter (g)

```
for (i = 1; i < m - 1; i++)
{
    f[i] = fv[i+1] +
           fv[i-1] -
           fv[i] * 2;
    cf2 = c * f[i] * f[i];
    dfv2 = d * fv[i] * fv[i];
    g[i] = cf2 * dfv2;
}
```

** Provided each thread has a private copy of the variable*

What Is Not Parallel?

Data dependencies (i.e., variables written in one iteration and read in another)

Subprograms with "state"

Data dependencies come in several flavors:

- Loop carried
- Induction variables
- Reductions
- Recurrence

Loop Carried Dependence

This loop is not parallel because `wrap` is carried from one iteration to the next

```
wrap = a[0] * b[0];
for (i = 1; i < N; i++)
{
    c[i] = wrap;
    wrap = a[i] * b[i];
    d[i] = 2 * wrap;
}
```

Restructure so `wrap` is defined before use in each iteration

```
for (i = 1; i < N; i++)
{
    wrap = a[i-1] * b[i-1];
    c[i] = wrap;
    wrap = a[i] * b[i];
    d[i] = 2 * wrap;
}
```

Induction Variables

Induction variables are incremented on each trip through the loop

```
i1 = 0;
i2 = 0;
for (i = 0; i < N; i++ )
{
    i1 += 1;
    B[i1] = ...
    i2 += i;
    A[i2] = ...
}
```

Fix by replacing increment expressions with pure function of loop index

```
for (i = 0; i < N; i++)
{
  B[i] = ...
  A[(i*i + i)/2] = ...
}
```

Reductions

Reductions collapse array data to scalar data via associative operations:

```
do i = 1, n
   sum = sum + c(i)
   maxx = max (maxx,c(i))
enddo
```

```
for (i = 0; i < n; i++)
    sum += c[i];
```

Take advantage of associativity and compute partial sums or local maxima in private storage

Next, combine partial results into shared result, taking care to synchronize access

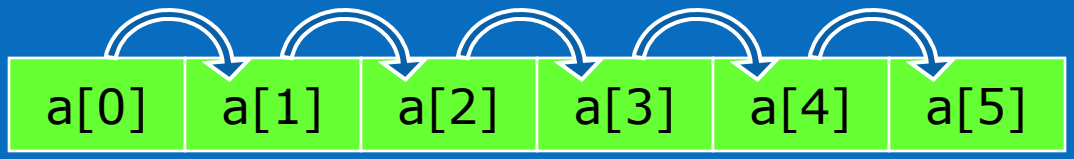
Recurrence

Recurrence relations feed information forward from one iteration to the next, e.g.

- Time stepping loops
- Convergence loops

```
for (i = 0; i < N; i++)
{
    a[i] = a[i-1] + b[i];
}
```

Unfortunately, most recurrences cannot be made parallel. Instead, look for a loop further out or in to parallelize.



Routines with State

Many routines maintain state across calls:

- Memory allocation
- Pseudo-random number generators
- I/O routines
- Graphics libraries
- Third-party libraries

Parallel access to such routines is unsafe unless synchronized

Check documentation for specific functions to determine thread-safety

A Simple Test

1. Reverse the loop order and rerun in serial
2. If results are unchanged, the loop is parallel*

**Exception: Loops with induction variables*

```
for (i = 0; i < N; i++)
{
   <...>
   compute (i, ...);
   <...>
}
```


```
for (i = N - 1; i >= 0; i--)
{
   <...>
   compute (i, ...);
   <...>
}
```

Agenda

Parallel Computing

An Introduction to OpenMP

Parallel Programming Tips and Techniques

Shared-Memory Parallelism

Multiple threads:

- Executing concurrently
- Sharing a single address space
- Sharing work in coordinated fashion

Scheduling handled by OS

Requires a system that provides shared memory and multiple processors

What Is OpenMP?

Portable, shared-memory threading API
- Fortran, C, and C++
- Multi-vendor support for both Linux and Windows

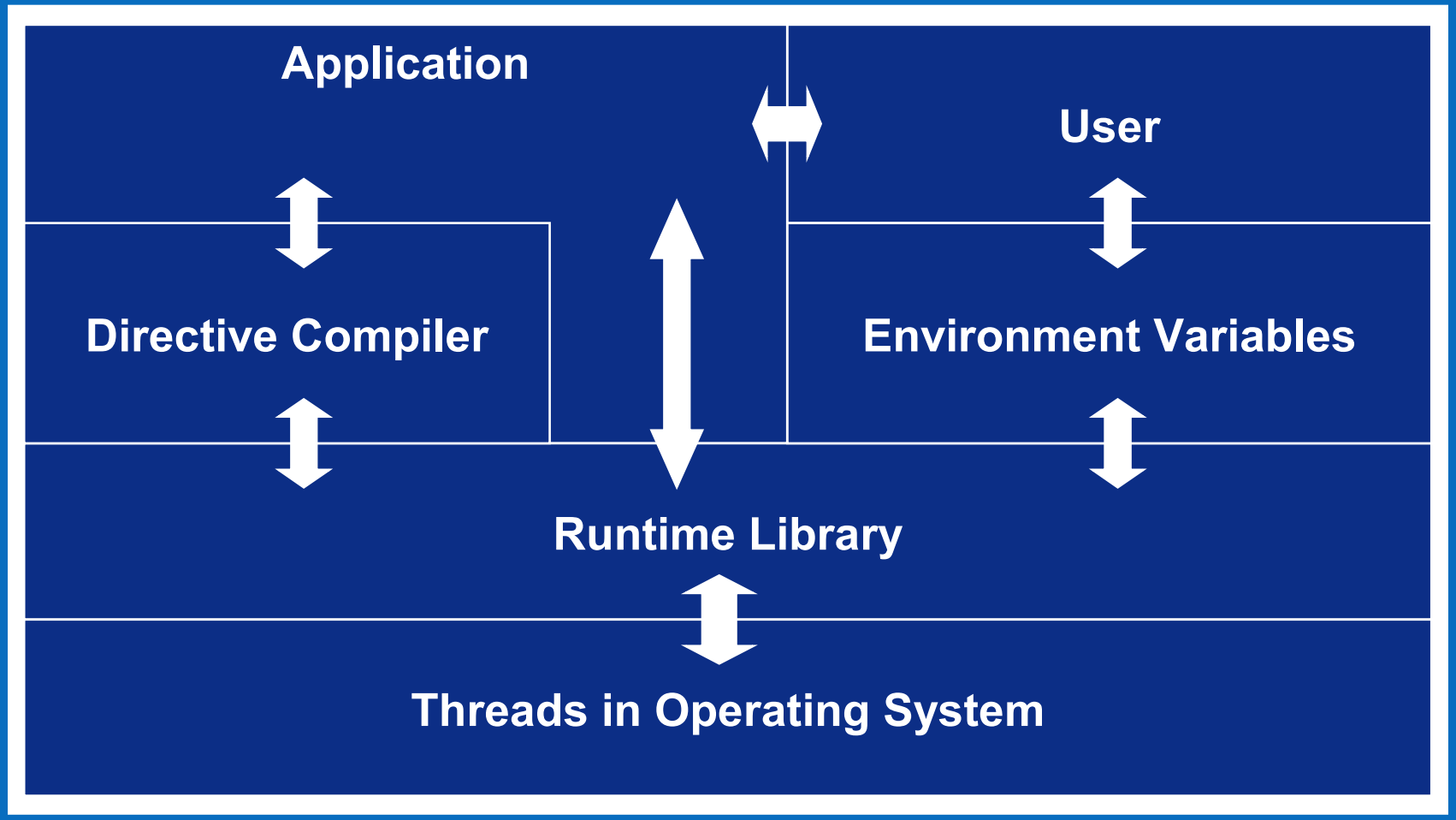
Standardizes loop-level parallelism

Supports coarse-grained parallelism

Combines serial and parallel code in single source

Standardizes almost 20 years of compiler-directed threading experience

OpenMP Runtime



Architecture

Fork-join model

Worksharing constructs

Synchronization constructs

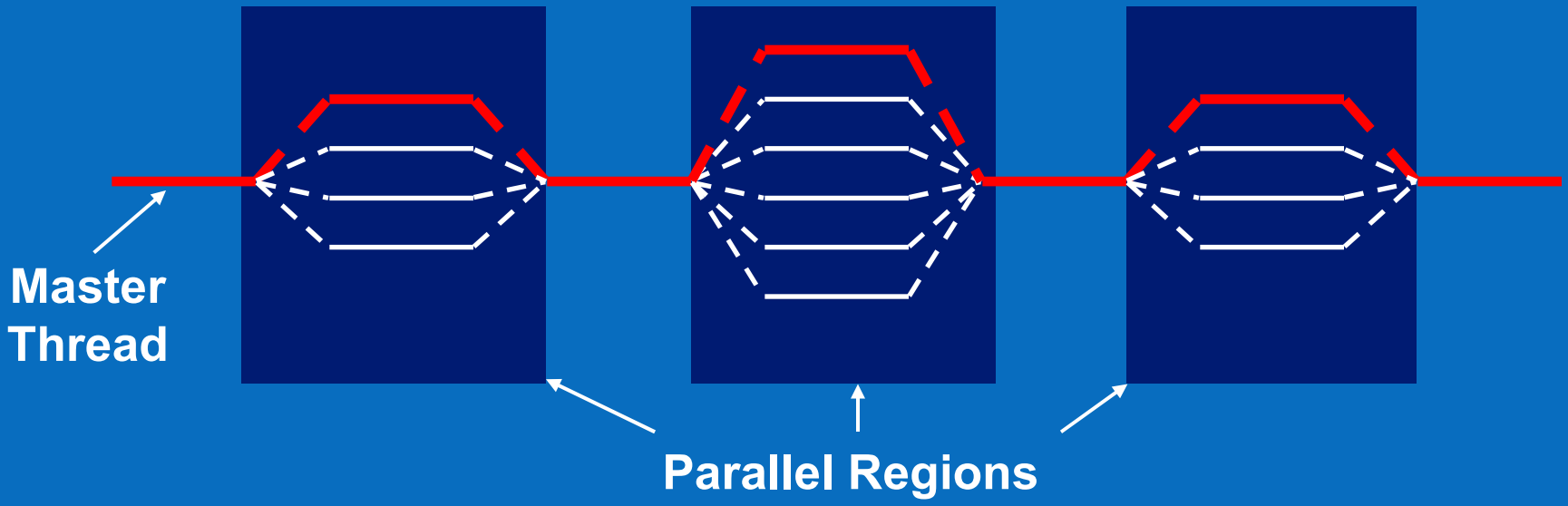
Directive/pragma-based parallelism

Extensive API library for better runtime control

Programming Model

Fork-Join Parallelism:

- Master thread spawns a team of threads as needed
- Parallelism is added incrementally: that is, the sequential program evolves into a parallel program



OpenMP Contents

OpenMP constructs fall into five categories

- Runtime functions/environment variables
- Parallel regions
- Worksharing
- Data environment
- Synchronization

OpenMP is fundamentally the same between Fortran and C/C++

A Few Syntax Details to Get Started

Most of the constructs in OpenMP are compiler directives or pragmas

- For C and C++, the pragmas take the form:

```
#pragma omp construct [clause [clause]…]
```

- For Fortran, the directives take one of the forms:

```
C$OMP construct [clause [clause]…]
!$OMP construct [clause [clause]…]
*$OMP construct [clause [clause]…]
```

Header file or Fortran 90 module

```
#include "omp.h"
use omp_lib
```

24 Library Routines

Runtime environment routines:

- Modify/check the number of threads

  `omp_[set|get]_num_threads()`

  `omp_get_thread_num()`

  `omp_get_max_threads()`
- Are we in a parallel region?

  `omp_in_parallel()`
- How many processors in the system?

  `omp_num_procs()`
- Explicit locks

  `omp_[set|unset]_lock()`
- And many more...

Library Routines

To fix the number of threads used in a program

- Set the number of threads
- Then save the number returned

```
#include <omp.h>

void main ()
{
   int num_threads;
   omp_set_num_threads (omp_num_procs ());

   #pragma omp parallel
   {
      int id = omp_get_thread_num ();

      #pragma omp single
      num_threads = omp_get_num_threads ();

      do_lots_of_stuff (id);
   }
}
```

Request as many threads as you have processors.

Protect this operation because memory stores are not atomic

Environment Variables

Set the default number of threads

```
OMP_NUM_THREADS integer
```

Set the default scheduling protocol

```
OMP_SCHEDULE "schedule[, chunk_size]"
```

Enable dynamic thread adjustment

```
OMP_DYNAMIC [TRUE|FALSE]
```

Enable nested parallelism

```
OMP_NESTED [TRUE|FALSE]
```

OpenMP Contents

OpenMP constructs fall into five categories

- Runtime functions/environment variables
- Parallel regions
- Worksharing
- Data environment
- Synchronization

OpenMP is fundamentally the same between Fortran and C/C++

Structured Blocks (C/C++)

Most OpenMP constructs apply to structured blocks

- Structured block: a block with one point of entry at the top and one point of exit at the bottom
- The only "branches" allowed are STOP statements in Fortran and exit() in C/C++

```
#pragma omp parallel
{
  int id = omp_get_thread_num();

more: res[id] = do_big_job (id);

  if (conv (res[id]) goto more;
}
printf (" All done \n");
```

A structured block

```
if (go_now()) goto more;
#pragma omp parallel
{
  int id = omp_get_thread_num();
more:  res[id] = do_big_job(id);
  if (conv (res[id]) goto done;
  goto more;
}
done: if (!really_done()) goto more;
```

Not a structured block

Structured Block Boundaries

In C/C++, a block is a single statement or a group of statements between brackets {}

```
#pragma omp parallel
{
  id = omp_thread_num ();
  res[id] = lots_of_work (id);
}
```

```
#pragma omp for
   for (i = 0; i < N; i++) {
     res[i] = big_calc (i);
     A[i] = B[i] + res[i];
   }
```

In Fortran, a block is a single statement or a group of statements between directive/end-directive pairs

```
C$OMP PARALLEL
10    wrk(id) = garbage(id)
      res(id) = wrk(id)**2
      if (conv(res(id)) goto 10
C$OMP END PARALLEL
```

```
C$OMP PARALLEL DO
   do I = 1, N
   res(I) = bigComp(I)
   end do
C$OMP END PARALLEL DO
```

Lab 1:
OpenMP "Hello World"

OpenMP Contents

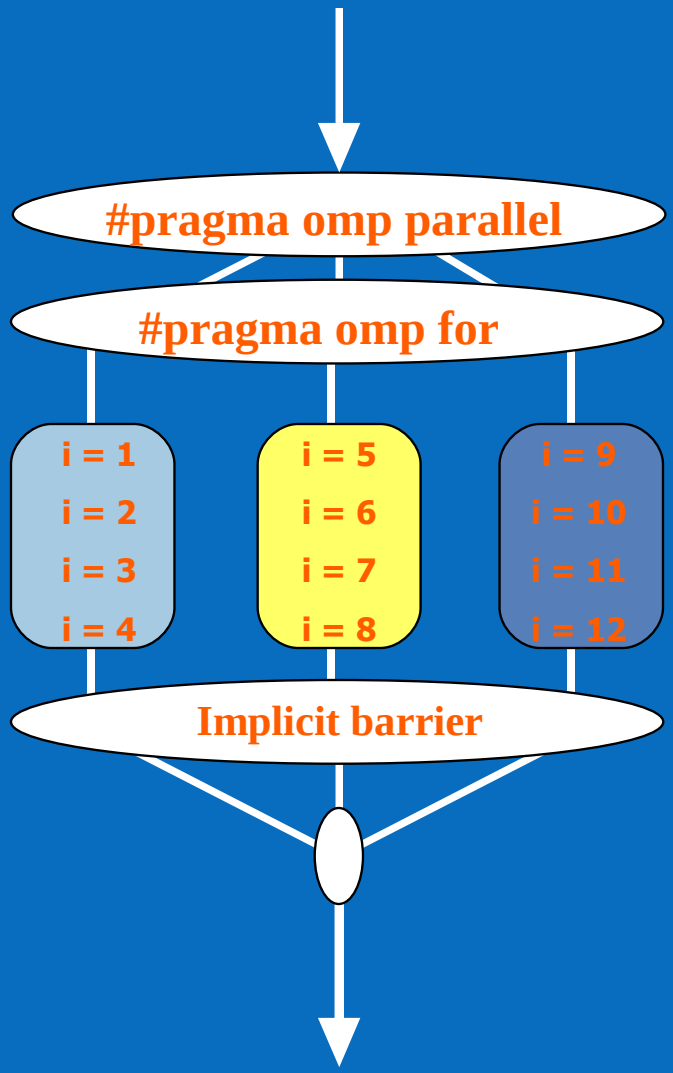
OpenMP constructs fall into five categories:

- Runtime functions/environment variables
- Parallel regions
- Worksharing
- Data environment
- Synchronization

OpenMP is fundamentally the same between Fortran and C/C++

Parallel Loops

```
#pragma omp parallel
{
    #pragma omp for
    for (i = 1; i < 13; i++)
    {
        c[i] = a[i] + b[i];
    }
}
```



Threads are assigned a set of independent iterations

Threads must wait at the end of a work-sharing construct

Parallel Loops

Find the most time consuming loops and divide the work

Divide this loop among multiple threads

```
int main ()
{
  int i;
  double Res[1000];

  for (i = 0; i < 1000; i++)
    do_huge_comp (Res[i]);
}
```

Sequential Program

```
int main ()
{
  int i;
  double Res[1000];
  #pragma omp parallel for
  for (i = 0; i < 1000; i++)
    do_huge_comp (Res[i]);
}
```

Parallel Program

Worksharing Constructs
A Motivating Example

Sequential code

```
for (i = 0; i < N; i++) a[i] = a[i] + b[i];
```

OpenMP parallel region

```
#pragma omp parallel
{
   int id, i, Nthrds, istart, iend;

   id = omp_get_thread_num();
   Nthrds = omp_get_num_threads();
   istart = id * N / Nthrds;
   iend = (id + 1) * N / Nthrds;

   for (i = istart; i < iend; i++)
      a[i] = a[i] + b[i];
}
```

OpenMP worksharing construct

```
#pragma omp parallel
#pragma omp for schedule(static)
   for (i = 0; i < N; i++) a[i] = a[i] + b[i];
```

The Schedule Clause

The schedule clause effects how loop iterations are distributed among threads

- schedule (static [,chunk])
  - Statically divide blocks of iterations of size "chunk" to each thread
- schedule (dynamic [,chunk])
  - Threads grab "chunk" iterations from a queue the moment they are ready to accept work
- schedule (guided [,chunk])
  - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size "chunk" as the calculation proceeds
- schedule (runtime)
  - Schedule and chunk size taken from the OMP_SCHEDULE environment variable

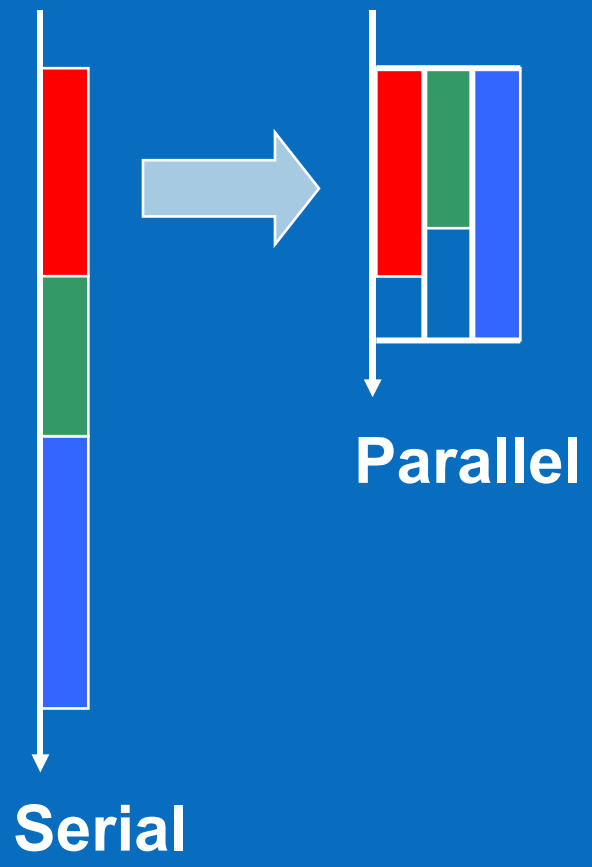
Recommended Scheduling

Schedule Clause	When To Use
STATIC	Predictable and approximately equal work per iteration
DYNAMIC	Unpredictable, highly variable work per iteration
GUIDED	Special case of dynamic scheduling for diminishing work per iteration
RUNTIME	Useful during tuning to find best scheduling protocol

OpenMP Parallel Sections

Specify blocks of code to execute in parallel

```
#pragma omp parallel sections
{
    #pragma omp section
    task1();
    #pragma omp section
    task2();
    #pragma omp section
    task3();
}
```



Combined Parallel and Worksharing Directives

```
!$OMP PARALLEL
!$OMP DO
      DO I = 1, N
         res(I) = huge()
      END DO
!$OMP END DO
!$OMP END PARALLEL
```

```
!$OMP PARALLEL DO
      DO I = 1, N
res(I) = huge()
      END DO
!$OMP END PARALLEL DO
```

These codes are equivalent

```
double  res[MAX];
int i;
#pragma omp parallel
{
#pragma omp for
   for (i = 0; i < MAX; i++)
      res[i] = huge();
}
```

```
double  res[MAX];
int i;

#pragma omp parallel for
for (i = 0; i < MAX; i++)
   res[i] = huge();
```

OpenMP Contents

OpenMP constructs fall into 5 categories:

- Runtime functions/environment variables
- Parallel regions
- Worksharing
- Data environment
- Synchronization

OpenMP is fundamentally the same between Fortran and C/C++

Scope of OpenMP Constructs

OpenMP parallel regions can span multiple source files

File: foo.f

```
C$OMP PARALLEL
   call whoami
C$OMP END PARALLEL
```

Lexical extent of parallel region

Dynamic extent of parallel region includes *lexical* extent

File: bar.f

```
   subroutine whoami
   external omp_get_thread_num
   integer iam, omp_get_thread_num
   iam = omp_get_thread_num()
C$OMP CRITICAL
   print*, 'Hello from ', iam
C$OMP END CRITICAL
   return
   end
```

Orphan directives can appear outside a parallel region

Data Environment: Default Storage

Shared-memory programming model

- Most variables are shared by default

Global variables are *shared* among threads

- Fortran: COMMON blocks, SAVE variables, MODULE variables
- C: File scope, namespace scope, and static variables

But not everything is shared...

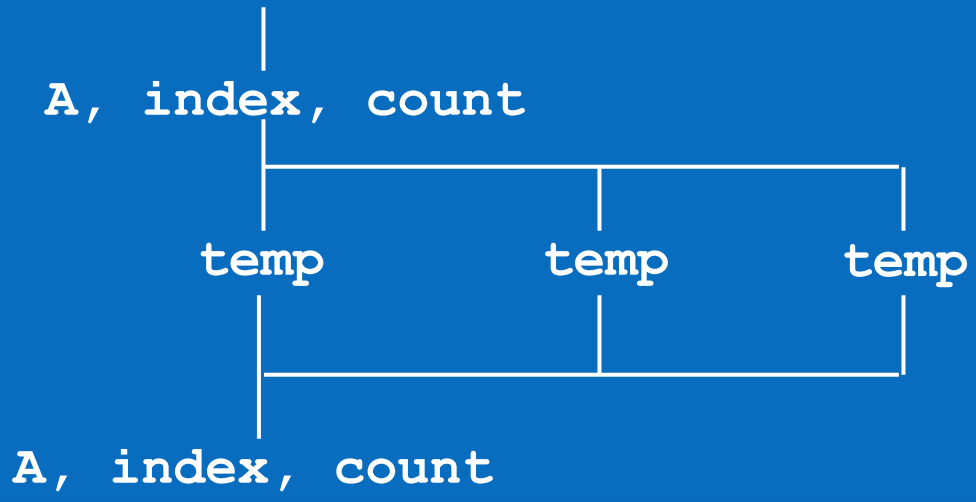
- Stack variables in subprograms called from parallel regions are *private*
- Automatic variables within a statement block are *private*

A Data Environment Example

```
float A[10];
main ()
{
   integer index[10];
   #pragma omp parallel
   {
      Work (index);
   }
   printf ("%d\n", index[1]);
}
```

```
extern float A[10];
void Work (int *index)
{
   float temp[10];
   static integer count;
   <...>
}
```

A, ***index***, **and** ***count*** **are shared by all threads, but** ***temp*** **is local to each thread**



Example: Parallel Dot Product

```
float DotProduct (float *a, float *b, int N)
{
    float sum = 0.0;

    #pragma omp parallel for
    for (int i = 0; i < N; i++)
    {
        sum += a[i] * b[i];
    }
    return sum;
}
```

Why does this code give wrong results?

Data Environment: Changing Storage Attributes

The scope of variables inside parallel regions can be changed using the following attribute clauses:

- SHARED
- PRIVATE
- FIRSTPRIVATE
- LASTPRIVATE
- REDUCTION
- THREADPRIVATE

The default attribute can be modified with:

- DEFAULT (PRIVATE | SHARED | NONE)

Default Clause

The default storage attribute is `DEFAULT(SHARED)` so no need to specify

To change default:

- `DEFAULT(PRIVATE)`
  - *Each* variable in *static* extent of the parallel region is made private as if specified in a private clause
  - Mostly saves typing when there are fewer shared variables
  - Not supported in C/C++
- `DEFAULT(NONE)`
  - *No* default for variables in static extent
  - Must list storage attribute for all variables
  - Sometimes useful during debugging

Private Clause

The `private(var)` clause creates a local copy of `var` for each thread

- The value is uninitialized in the parallel region
- Private copy is *not* storage associated with the original
- Loop indices are private by default

```
#pragma omp parallel for private (cf2, dfv2)
for (i = 1; i < N - 1; i++)
{
   f[i] = fv[i+1] + fv[i-1] – fv[i] * 2;
   cf2 = c * f[i] * f[i];
   dfv2 = d * fv[i] * fv[i];
   A[i] = cf2 * dfv2;
}
```

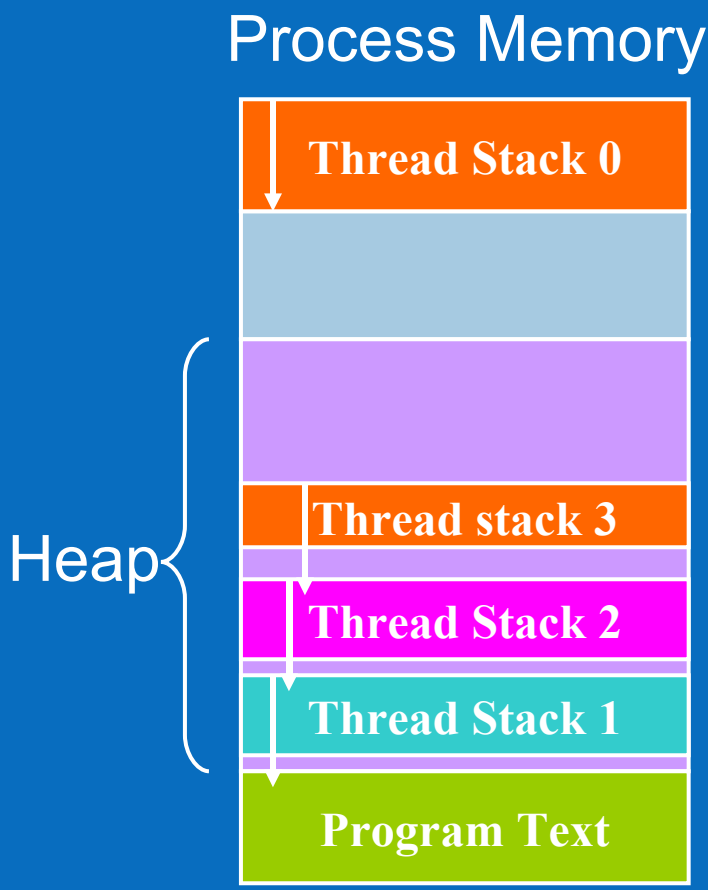
Stack Collisions

All worker thread stacks are allocated out of a common memory heap

Large private data and deep call trees can cause stack corruption

Intel implementation:

- Default stack size
  - IA-32: 2 MB
  - Itanium: 4 MB
- Change stack size
  - KMP_STACKSIZE
  - kmp_set_stacksize_s



Firstprivate Clause

Firstprivate is a special case of private. It initializes each private copy with the corresponding value from the master thread.

```
int j = 5;

#pragma omp parallel for firstprivate (j)
for (i = 0; i < N; i++)
{
   A[i] = j * i;
}
```

Lastprivate Clause

Lastprivate is a special case of private. It passes the value of a private from the last iteration to a global variable.

```
int i = 0;

#pragma omp parallel for lastprivate (i)
for (i = 0; i < N - 1; i++)
{
   A[i] = B[i] + B[i+1];
}
A[i] = B[i];    /* i == (N-1) */
```

Example code taken from the *OpenMP Application Program Interface*, v2.5.

Reduction Clause

Reduction is a special case of shared

- reduction (op: list)
- Variables in "list" must be shared in the enclosing parallel region

Inside a parallel region or worksharing construct

- A local copy of each reduction variable is made and initialized depending on the "op" (e.g., 0 for "+")
- Compiler finds standard reduction expressions containing "op" and uses them to update the local copy
- Local copies are reduced to a single value and combined with the original global value

Reduction Operands and Initial Values

A range of associative operands can be used with reduction

Initial values are the ones that make sense mathematically

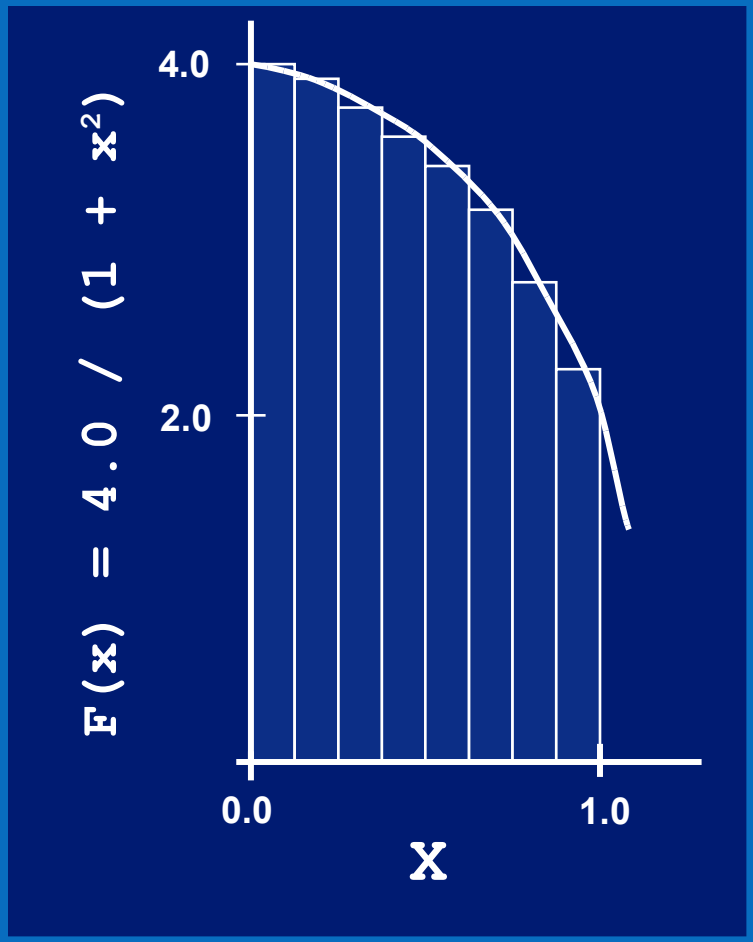
Operand	Initial value
+	0
*	1
-	0
.AND.	All 1's

Operand	Initial value
.OR.	0
MAX	-∞
MIN	+∞

Reduction Example

```
int i, sum;

sum = 0;

#pragma omp parallel for reduction (+:sum)
for (i = 0; i < N; i++)
{
    sum += A[i];
}

printf ("Sum = %d\n", sum);
```

Computing Pi by Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1 + x^2)} \, dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i) \, \Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

The Sequential Pi Program

```
static int num_steps = 100000;
double step;

int main () {
    int i;
    double x, pi, sum = 0.0;

    step = 1.0 / (double) num_steps;
    for (i = 0; i < num_steps; i++)
    {
        x = (i + 0.5) * step;
        sum = sum + 4.0 / (1.0 + x * x);
    }
    pi = step * sum;
    return 0;
}
```

Lab 2:
Multithreaded Pi Integration

Lab 3 (Optional):
Manual Debugging

Contents

OpenMP's constructs fall into five categories:

- Runtime functions/environment variables
- Parallel regions
- Worksharing
- Data environment
- Synchronization

OpenMP is fundamentally the same between Fortran and C/C++

Synchronization

OpenMP has the following synchronization constructs:

- critical
- atomic
- barrier
- flush
- ordered
- master
- single

We will discuss the OpenMP single construct in terms of synchronization even though it is a worksharing construct.

OpenMP also provides explicit locks

- omp_init_lock, omp_set_lock, omp_unset_lock

OpenMP Critical Sections

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
   int type = getType (i);
   double force = computeForce (i);
   switch (type%2) {
      case 0:
         #pragma omp critical
         totForce[type] += force;
         break;
      case 1:
         #pragma omp critical
         totForce[type] += force;
         break;
   }
}
```

Only one thread at a time can enter a critical section

The critical section in this example protects totForce from concurrent updates

Named Critical Sections

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
   int type = getType (i);
   double force = computeForce (i);
   switch (type%2) {
      case 0:
         #pragma omp critical(mod0)
         totForce[type] += force;
         break;
      case 1:
         #pragma omp critical(mod1)
         totForce[type] += force;
         break;
   }
}
```

Named critical sections can be used to protect unrelated code or data, and reduce lock contention

In this example, the updates in ‘case 0’ and ‘case 1’ do not conflict

OpenMP Ordered Construct

The ordered construct enforces the sequential order for a block

```
#pragma omp parallel for private(tmp) ordered
   for (i = 0; i < N; i++)
   {
      tmp = big_func ();
      #pragma omp ordered
      {
         res += consume (tmp);
      }
   }
```

The performance penalty for imposing sequential order is usually severe

Atomic Updates

Atomic is a special critical section that only applies to the update of a memory location:

```
#pragma omp parallel
{
    temp = big_func ();
    #pragma omp atomic
    x += temp;
}
```

Atomic updates are faster than OpenMP critical sections or explicit locks

Barriers

```
#pragma omp parallel shared (A, B, C) private (id)
{
   id = omp_get_thread_num ();
   A[id] = big_calc1 (id);
   #pragma omp barrier

   #pragma omp for
   for (i = 0; i < N; i++)
      C[i] = big_calc3 (i, A);

   #pragma omp for nowait
   for (i = 0; i < N; i++)
      B[i] = big_calc2 (C, i);

   A[id] = big_calc3 (id);
}
```

No thread may pass barrier until all threads arrive

Implicit barrier at the end of a worksharing construct

No implicit barrier due to nowait. Is this safe?

Implicit barrier at the end of a parallel region cannot be removed

OpenMP Master Construct

The master construct denotes a structured block that is only executed by the master thread

There is no implicit barrier on a master block

```
#pragma omp parallel
{
    do_many_things ();
    #pragma omp master
    {
        exchange_boundaries ();
    }
    #pragma omp barrier
    do_many_other_things ();
}
```

OpenMP Single Construct

The single construct denotes a block of code in a parallel region that is executed by only one thread

There is an implicit barrier at the end of the single block (remove with nowait clause)

```
#pragma omp parallel
{
    do_many_things ();
    #pragma omp single
    {
        exchange_boundaries ();
    }
    do_many_other_things ();
}
```

Intel Extensions to OpenMP (Optional)

Work Queuing Model

Intel extension to OpenMP

Better handling of common programming idioms

- Recursive control
- List or tree traversal

Improves OpenMP task-parallel capability

Work queuing expected in future OpenMP specification

Work Queuing Example: Pointer Chasing

```
nodeptr list;
...
#pragma intel omp parallel taskq
for (nodeptr p = list; p != NULL; p = p->next)
{
   #pragma intel omp task
      process (p->data);
   #pragma intel omp task
      process (p->more_data);
}
```

Example code taken from Shah *et al*., "Flexible Control Structures for Parallelism in OpenMP" *Concurrency: Practice and Experience*, 12:1219-1239, 2000

High-Performance Computing Clusters

Workstation clusters have become the *de facto* HPC platform

MPI is the most common parallel programming method for HPC but it is hard to use

Intel has extended OpenMP for use on distributed-memory, parallel computers

Cluster OpenMP

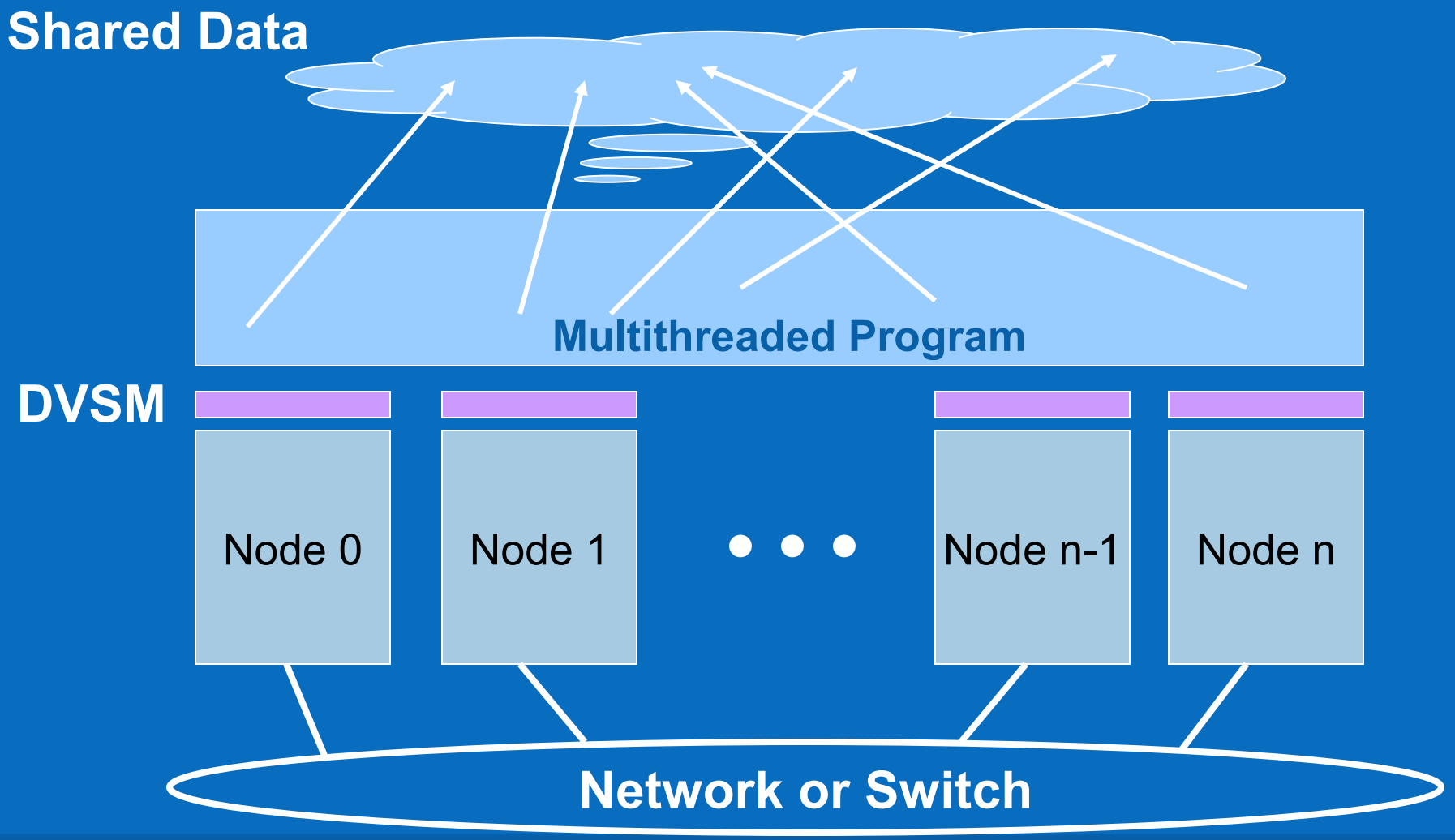
Cluster OpenMP allows OpenMP programs to run on clusters

One additional directive (`sharable`) extends OpenMP

The `sharable` directive designates variables that are shared by OpenMP threads

The compiler can determine some variables as *sharable* at compile time but the user must use directives for the rest

Distributed Virtual Shared Memory



DVSM Protocol

Basic idea:

- When a page of sharable memory is out-of-date, it is *protected* and an access to it faults into our software, which requests info from remote nodes to bring it up-to-date.
- Protection is removed from page
- Instruction causing the fault is restarted, this time successfully accessing the data

Summary

OpenMP best suited to data parallelism

OpenMP does not guarantee correctness

OpenMP does not guarantee performance

Intel Threading Tools make threading easier

- Use Intel Thread Checker to debug
- Use Intel Thread Profiler to tune


intel®
Leap ahead™

Backup

Agenda

Parallel Programming Tips and Techniques

- Parallel debugging
- Parallel tuning

How Do Threads Interact?

OpenMP is a shared-memory model

Threads communicate by sharing variables

Race conditions occur because a particular order of execution is assumed but not guaranteed

- Storage conflict: A race condition involving unsynchronized access to shared data

To avoid race conditions

- Use synchronization to enforce execution order
- Use synchronization to protect shared data

Synchronization is expensive so it is better to change how data are accessed to minimize the need for synchronization

Common Parallel Bugs

What can go wrong?

- Incorrectly classified variables
- Unsynchronized access to shared variables
  - Data read before written (Read/Write conflict)
  - Data overwritten before read (Write/Read conflict)
  - Simultaneously writes (Write/Write conflict)
- Uninitialized private data
- Failure to update global data
- Timing-dependent bugs

Common Parallel Bugs

What else can go wrong?

- Unsynchronized access to I/O or thread-unsafe libraries
- Thread stack collisions
- Inconsistent `threadprivate` variable declarations

Debugging Tips

Check parallel P=1 results

If results differ from serial, check for:

- Uninitialized private data
- Missing lastprivate clauses

If results are same as serial, check for:

- Unsynchronized access to shared variables (including static and SAVE variables)
- Shared variables that should be private
- Inconsistent `threadprivate` variable declarations

OpenMP Deadlock

In this example, deadlock occurs because threads arrive at different barriers

If one thread skips a barrier, it generally causes deadlock

Nested critical sections or locks can cause deadlock

```
#pragma omp parallel
private(me)
{
    int me;
    me = omp_get_thread_num ();
    if (me == 0) goto Master;
    #pragma omp barrier

Master:

    #pragma omp single
    write(*,*) "done"
}
```

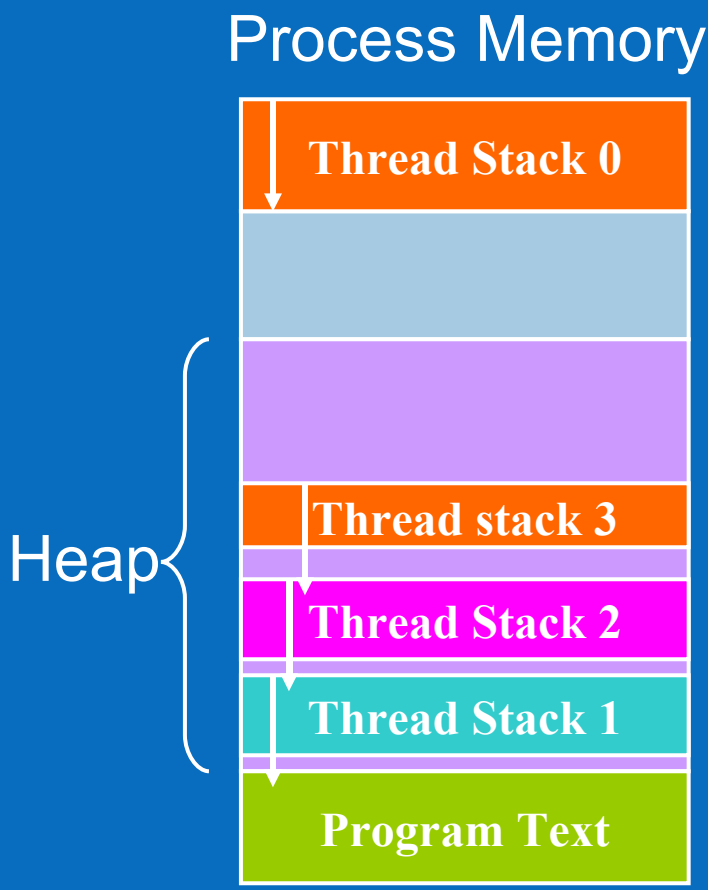
Stack Collisions

All worker thread stacks are allocated out of a common memory heap

Large private data and deep call trees can cause stack corruption

Intel implementation:

- Default stack size
  - IA-32: 2 MB
  - Itanium: 4 MB
- Change stack size
  - KMP_STACKSIZE
  - kmp_set_stacksize_s



Parallel Debugging is Hard

Parallel Bugs:

- Often cross function boundaries
- Are often timing dependent
- Can be intermittent
- Are sometimes nearly imperceptible
- Are sometimes difficult to reproduce

Debugging probes can mask bugs

Debugging Options

Print statements

Multithreaded debuggers

Intel Thread Checker

Print Statements

Advantages

- WYSIWYG
- Can be useful for deterministic bugs
- Can monitor scheduling of iterations on threads

Disadvantages

- Slow, human-intensive bug hunting
- Can make the bug "go away"

Tips

- Include thread ID in messages
- Compute and print checksums on shared memory regions
- Protect I/O with a `critical` section

Multithreaded Debugger

Advantages

- Can find symptoms of deadlock, such as threads waiting at different barriers

Disadvantages

- Locates symptom, not cause
- Hard to reproduce errors, especially those that are timing-dependent
- Difficult to relate parallel library calls back to original source
- Human-intensive

Intel Thread Checker

Finds errors in multithreaded code, e.g.

- Data races
- Deadlocks and thread stalls
- Invalid API calls
- Memory problems

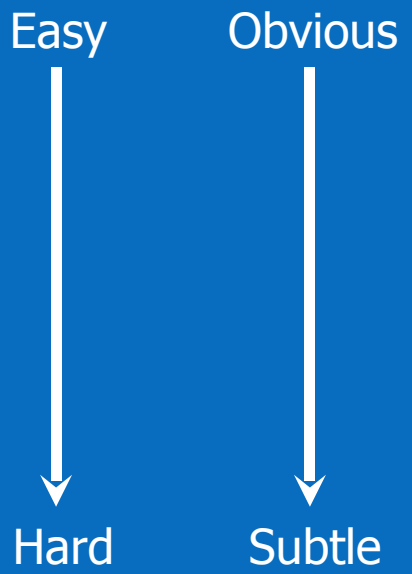
Supports OpenMP, Pthreads, Windows threading API

Agenda

Parallel Programming Tips and Techniques

- Parallel debugging
- Parallel tuning

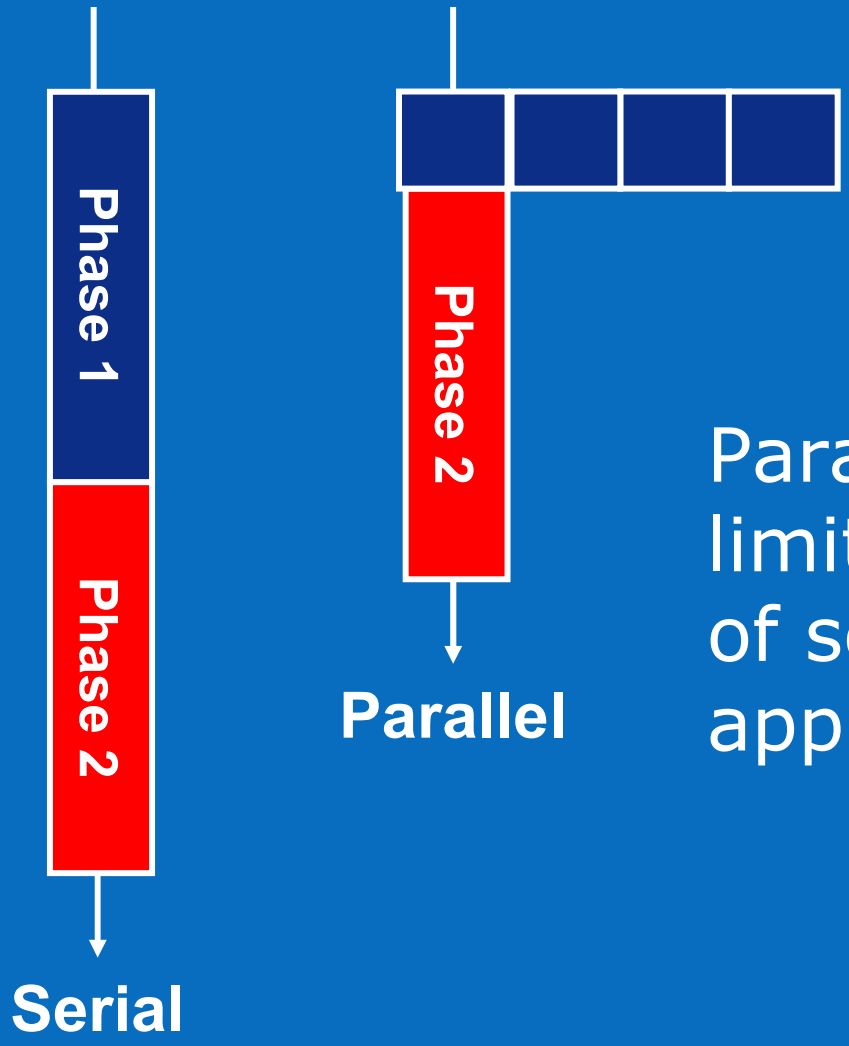
Parallel Performance



Limiters of performance

- Amdahl's law
- Load imbalance
- Communication
- Synchronization
- Overheads
- False sharing

Amdahl's Law

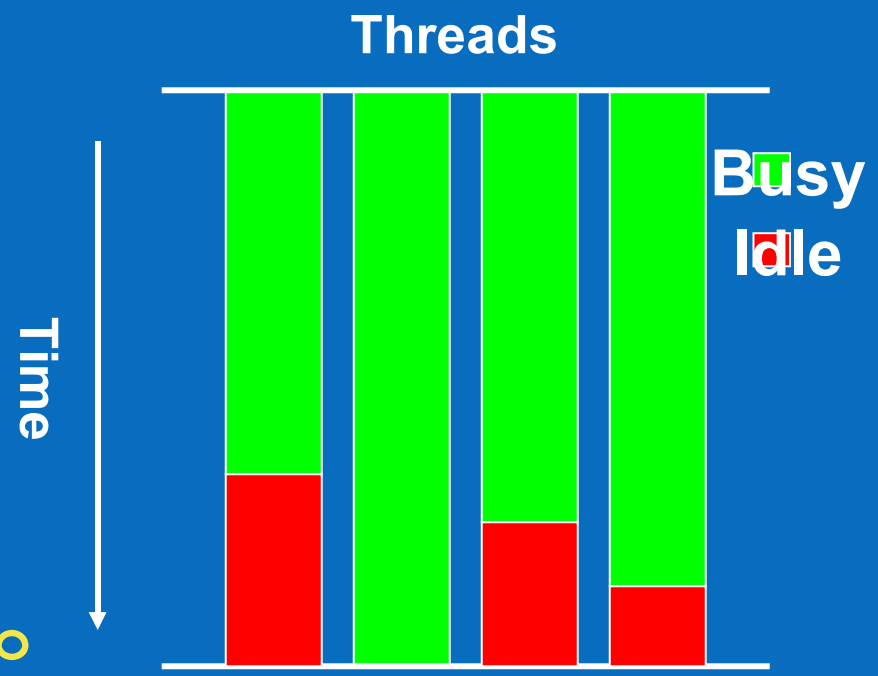


Parallel speedup is limited by the amount of serial code in an application.

Load Balance

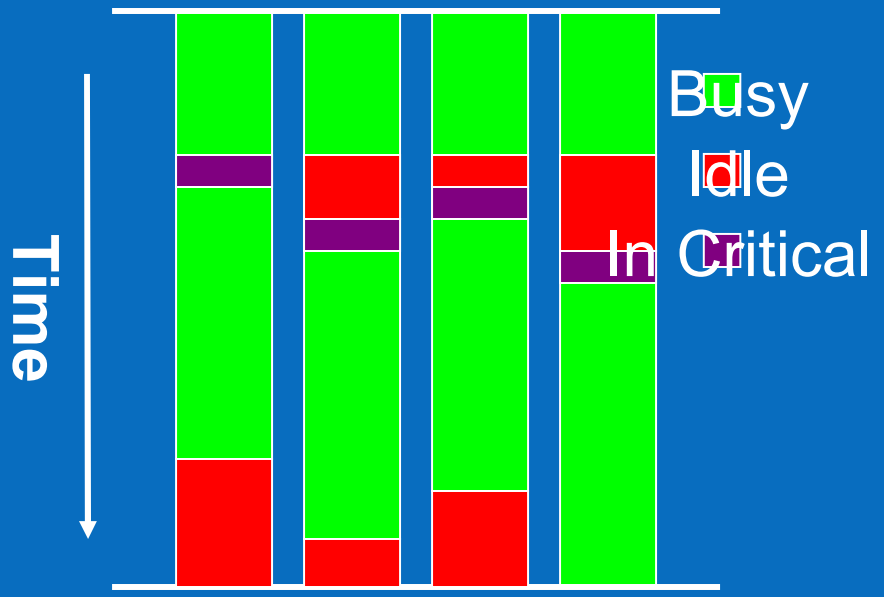
Unequal work leads to idle threads and reduces parallel efficiency and scalability

```
!$omp parallel do

!$omp end parallel do
```



Synchronization

```
#pragma omp parallel
{

#pragma omp critical
   {
     ...
   }
}
```



Tuning Synchronization

Reduce lock contention

- Different named critical sections
- Introduce domain-specific locks

Replace critical sections and locks with atomic updates when possible

Merge parallel loops and remove barriers

Merge small critical sections

Move critical sections outside loops

Limiting Parallel Overheads

Merge adjacent parallel regions

When safe, remove implicit barrier at end of worksharing constructs

Eliminate small parallel loops

- Use `if` clause to limit parallelism to large loops
- Increase problem size

Adjust `KMP_BLOCKTIME`

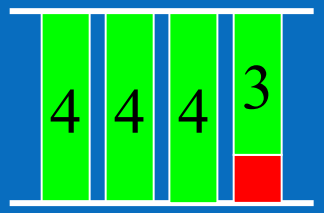
Small Loops

Small trip counts can cause load imbalance

Increase problem size

Merge parallel loop with inner loop to expose more parallelism

```
#pragma omp for
   for (i = 0; i < 15; i++)
      {...}
```




```
do i = 1, imax
   do j = 1, jmax
      [...]
enddo

do ij= 1, imax*jmax
   i = (ij-1) / jmax + 1
   j = ij - (i-1) * jmax
   [...]
enddo
```

False Sharing

False sharing occurs when multiple threads repeatedly write to the same cache line

```
#define MAX_THREADS 16
double sum = 0.0, sum_local[MAX_THREADS];
#pragma omp parallel
{
    int me = omp_get_thread_num();
    sum_local[me] = 0.0;

    #pragma omp for
    for (i = 0; i < N; i++)
        sum_local[me] += x[i] * y[i];

    #pragma omp atomic
    sum += sum_local[me];
}
```

Intel Thread Profiler

Finds OpenMP-specific performance bottlenecks

- Load imbalance
- Lock contention
- Parallel overhead

Agenda

Advanced OpenMP Constructs

Threadprivate

Makes global data private to a thread

- Fortran: Named `COMMON` blocks, module, and `SAVE` variables
- C: File scope, namespace scope, and static variables

Different from making variables `PRIVATE`

- `PRIVATE` masks global variables
- `THREADPRIVATE` preserves global scope within each thread

Threadprivate variables can be initialized using the `COPYIN` clause

Threadprivate Example

```
float *work;
int size;
float tol;
#pragma omp threadprivate (work, size, tol)

void a32 (float t, int n)
{
   tol = t; size = n;
   #pragma omp parallel copyin (tol, size)
   {
      build ();
   }
}

void build ()
{
   work = (float *) malloc (sizeof (float) * size);
   for (int i = 0; i < size; ++i) work[i] = tol;
}
```

The threadprivate clause creates private copies of global data for each thread

The copyin clause copies the master's threadprivate contents to the worker threads

Example code taken from the *OpenMP Application Program Interface*, v2.5.

Copyprivate

The `copyprivate` clause broadcasts private variables from one thread to all other threads

```
float x, y;
#pragma omp threadprivate (x, y)

void InitVars (float a, float b)
{
   #pragma omp single copyprivate (a, b, x, y)
   {
      scanf ("%f %f %f %f", &a, &b, &x, &y);
   }
}
```

Note that a and b must be private if InitVars is called from a parallel region.

The thread that reads variables a, b, x, and y broadcasts their values to the other threads.

Example code taken from the *OpenMP Application Program Interface*, v2.5.

OpenMP Explicit Locks

```
#include <omp.h>
<...>
omp_lock_t lock;
omp_init_lock (&lock);

#pragma omp parallel for
for (i = 0; i < N; i++)
{
    int type = getType (i);
    double force = computeForce (i);
    omp_set_lock (&lock);
    totForce[type] += force;
    omp_unset_lock (&lock);
}
```

Explicit locks give finer control over synchronization, e.g.:

- Associate a lock with individual elements of a data structure
- Nested locks

OpenMP explicit locks are similar to Windows and Pthreads mutex variables